8. Context-Based Coding

In this section we present a number of techniques that use minimal prior assumptions about the statistics of the data. Instead they use the context of the data being encoded and the past history of the data to provide more efficient compression. We will look at a number of schemes that are principally used for the compression of text. These schemes use the context in which the data occurs in different ways.

8.1 Prediction with Partial Match (PPM)

The best-known context-based algorithm is the ppm algorithm, first proposed by Cleary and Witten in 1984. It has not been as popular as the various Ziv-Lempel-based algorithms mainly because of the faster execution speeds of the latter algorithms. Lately, with the development of more efficient variants, PPM-based algorithms are becoming increasingly more popular.

The idea of the PPM algorithm is elegantly simple. We would like to use large contexts to determine the probability of the symbol being encoded. However, the use of large contexts would require us to estimate and store an extremely large number of conditional probabilities, which might not be feasible. Instead of estimating these probabilities ahead of time, we can reduce the burden by estimating the probabilities as the coding proceeds. This way we only need to store those contexts that have occurred in the sequence being encoded. This is a much smaller number than the number of all possible contexts. While this mitigates the problem of storage, it also means that, especially at the beginning of an encoding, we will need to code letters that have not occurred previously in this context. In order to handle this situation, the source coder alphabet always contains an escape symbol, which is used to signal that the letter to be encoded has not been seen in this context.

The basic algorithm initially attempts to use the largest context. The size of the largest context is predetermined. If the symbol to be encoded has not previously been encountered in this context, an escape symbol is encoded, and the algorithm attempts to use the next smaller context. If the symbol has not occurred in this context either, the size of the context is further reduced. This process continues until either we obtain a context that has previously been encountered with this symbol, or we arrive at the conclusion that the symbol has not been encountered previously in any context. In this case, we use a probability of 1/M to encode the symbol, where M is the size of the source alphabet. For example, when coding the *a* of *probability*, we would first attempt to see if the string *proba* has previously occurred— that is, if *a* had previously occurred in the context of *prob*. If not, we would encode an escape and see if *a* had occurred in the context of *rob*.

If the string *roba* had not occurred previously, we would again send an escape symbol and try the context *ob*.

Continuing in this manner, we would try the context *b*, and failing that, we would see if the letter *a* (with a zero-order context) had occurred previously. If *a* was being encountered for the first time, we would use a model in which all letters occur with equal probability to encode *a*. This equiprobable model is sometimes referred to as the context of order -1. As the development of the probabilities with respect to each context is an adaptive process, each time a symbol is encountered, the count corresponding to that symbol is updated.

The PPM Algorithm

- The maximal context order, N, must be selected in advance
- For $K = N$ .. 1
  - Read K symbols before current symbol s
  - If the K-th order context is not available, decrement K and continue
  - Otherwise, if the K-th order context does not contain s
    - Encode an escape symbol
    - Decrement K and continue
  - Otherwise, encode s using the K-th order context and break loop
- If s is not encoded, use -1 order statistics
  - All symbols have equal probabilities
- After encoding, update context statistics (not for escape code)

Example:

Let’s encode the sequence: ***thisb/isb/theb/tithe***

Assuming we have already encoded the initial seven characters ***thisb/is***, the various Counts and Cum_Count arrays to be used in the arithmetic coding of the symbols are shown in Tables 6.1–6.4. In this example, we are assuming that the longest context length is **two**. This is a rather small value and is used here to keep the size of the example reasonably small. A more common value for the longest context length is five.

TABLE 6.1 **Count array for −1 order context.**

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	1	3
s	1	4
e	1	5
b̸	1	6
Total Count		6

TABLE 6.2 **Count array for zero-order context.**

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	2	4
s	2	6
b̸	1	7
⟨*Esc*⟩	1	8
Total Count		8

TABLE 6.3 **Count array for first-order contexts.**

Context	Letter	Count	*Cum_Count*
t	*h*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
h	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
i	*s*	2	2
	⟨*Esc*⟩	1	3
	Total Count		3
b̸	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
s	*b̸*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2

TABLE 6.4 Count array for second-order contexts.

Context	Letter	Count	*Cum_Count*
th	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
hi	*s*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
is	*b/*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
sb/	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
b/i	*s*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2

We will assume that the word length for arithmetic coding is **six**. Thus, l = **000000** and u = 111111. As ***thisb/*is** has already been encoded, the next letter to be encoded is ***b/***. The second-order context for this letter is ***is***. Looking at Table 6.4, we can see that the letter ***b/*** is the first letter in this context with a Cum_Count value of 1. As the Total_Count in this case is 2, the update equations for the lower and upper limits are

$$l = 0 + \left\lfloor (63 - 0 + 1) \times \frac{0}{2} \right\rfloor = 0 = 000000$$

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{2} \right\rfloor - 1 = 31 = 011111.$$

As the MSBs of both ***l*** and ***u*** are the same, we shift that bit out, shift a 0 into the LSB of ***l***, and a 1 into the LSB of ***u***. The transmitted sequence, lower limit, and upper limit after the update are

Transmitted sequence: 0

l: 000000
u: 111111

We also update the counts in Tables 6.2–6.4. The next letter to be encoded in the sequence is ***t***. The second-order context is ***sb/***. Looking at Table 6.4, we can see that ***t*** has not appeared before in this context. We therefore encode an escape symbol. Using the counts listed in Table 6.4, we update the lower and upper limits:

$$l = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{2} \right\rfloor = 32 = 100000$$

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{2}{2} \right\rfloor - 1 = 63 = 111111.$$

Again, the MSBs of l and ***u*** are the same, so we shift the bit out and shift 0 into the LSB of l, and 1 into ***u***, restoring l to a value of 0 and u to a value of 63. The transmitted sequence is now 01. After transmitting the escape, we look at the first-order context of ***t***, which is ***b/***. Looking at Table 6.3, we can see that ***t*** has not previously occurred in this context. To let the decoder know this, we transmit another escape. Updating the limits, we get

$$l = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{2} \right\rfloor = 32 = 100000$$

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{2}{2} \right\rfloor - 1 = 63 = 111111.$$

As the MSBs of ***l*** and ***u*** are the same, we shift the MSB out and shift 0 into the LSB of ***l*** and 1 into the LSB of ***u***. The transmitted sequence is now 011. Having escaped out of the first-order contexts, we examine Table 6.5, the updated version of Table 6.2, to see if we can encode ***t*** using a zero-order context. Indeed we can, using the Cum_Count array, update ***l*** and ***u***:

$$l = 0 + \left\lfloor (63 - 0 + 1) \times \frac{0}{9} \right\rfloor = 0 = 000000$$

$$u = 0 + \left\lfloor (63 - 0 + 1) \times \frac{1}{9} \right\rfloor - 1 = 6 = 000110.$$

TABLE 6.5 **Updated count array for zero-order context.**

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	2	4
s	2	6
b̸	2	8
⟨*Esc*⟩	1	9
Total Count		9

The three most significant bits of both ***l*** and ***u*** are the same, so we shift them out. After the update we get

Transmitted sequence: 011000

l: 000000
u: 110111

The next letter to be encoded is ***h***. The second-order context ***b̸/t*** has not occurred previously, so we move directly to the first-order context ***t***. The letter ***h*** has occurred previously in this context, so we update ***l*** and ***u*** and obtain

Transmitted sequence: 0110000

l: 000000
u: 110101

The method of encoding should now be clear. At this point the various counts are as shown in Tables 6.6–6.8.

TABLE 6.6 **Count array for zero-order context.**

Letter	Count	*Cum_Count*
t	2	2
h	2	4
i	2	6
s	2	8
b̸	2	10
⟨*Esc*⟩	1	11
Total Count		11

TABLE 6.7 Count array for first-order contexts.

Context	Letter	Count	*Cum_Count*
t	*h*	2	2
	⟨*Esc*⟩	1	3
	Total Count		3
h	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
i	*s*	2	2
	⟨*Esc*⟩	1	3
	Total Count		3
b̸	*i*	1	1
	t	1	2
	⟨*Esc*⟩	1	3
	Total Count		3
s	*b̸*	2	2
	⟨*Esc*⟩	1	3
	Total Count		3

TABLE 6.8 Count array for second-order contexts.

Context	Letter	Count	*Cum_Count*
th	*i*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
hi	*s*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
is	*b̸*	2	2
	⟨*Esc*⟩	1	3
	Total Count		3
sb̸	*i*	1	1
	t	1	2
	⟨*Esc*⟩	1	3
	Total Count		3
b̸i	*s*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2
b̸t	*h*	1	1
	⟨*Esc*⟩	1	2
	Total Count		2

8.2 Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) algorithm also uses the context of the symbol being encoded, but in a very different way, for lossless compression. The transform that is a major part of this algorithm was developed by Wheeler in 1983. However, the BWT compression algorithm, which uses this transform, saw the light of day in 1994. Unlike most of the previous algorithms we have looked at, the BWT algorithm requires that the entire sequence to be coded be available to the encoder before the coding takes place.

The algorithm can be summarized as follows. Given a sequence of length **N**, we create **N − 1** other sequences where each of these **N − 1** sequences are a cyclic shift of the original sequence. These **N** sequences are arranged in lexicographic order. The encoder then transmits the sequence of length **N** created by taking the last letter of each sorted, cyclically shifted, sequence. This sequence of last letters **L**, and the position of the original sequence in the sorted list, are coded and sent to the decoder. As we shall see, this information is sufficient to recover the original sequence.

Example: Let’s encode the sequence ***thisb/isb/the***

We start with all the cyclic permutations of this sequence. As there are a total of 11 characters, there are 11 permutations, shown in Table 6.14.

TABLE 6.14 **Permutations of *thisb/isb/the*.**

0	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*
1	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*
2	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*
3	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*
4	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*
5	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*
6	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*
7	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*
8	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*
9	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*
10	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*

TABLE 6.15 Sequences sorted into lexicographic order.

0	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*
1	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*
2	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*
3	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*
4	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*
5	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*
6	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*
7	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*
8	*s*	*b/*	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*
9	*t*	*h*	*e*	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*
10	*t*	*h*	*i*	*s*	*b/*	*i*	*s*	*b/*	*t*	*h*	*e*

Now let's sort these sequences in lexicographic (dictionary) order (Table 6.15). The sequence of last letters L in this case is

L: sshtthb/iib/e

Notice how like letters have come together. If we had a longer sequence of letters, the runs of like letters would have been even longer. The *mtf* algorithm, which we will describe later, takes advantage of these runs. The original sequence appears as sequence number 10 in the sorted list, so the encoding of the sequence consists of the sequence **L** and the index value 10.

Now that we have an encoding of the sequence, let's see how we can decode the original sequence by using the sequence L and the index to the original sequence in the sorted list. The important thing to note is that all the elements of the initial sequence are contained in L. We just need to figure out the permutation that will let us recover the original sequence.

The first step in obtaining the permutation is to generate the sequence F consisting of the first element of each row. That is simple to do because we lexicographically ordered the sequences. Therefore, the sequence F is simply the sequence L in lexicographic order. In our example this means that F is given as

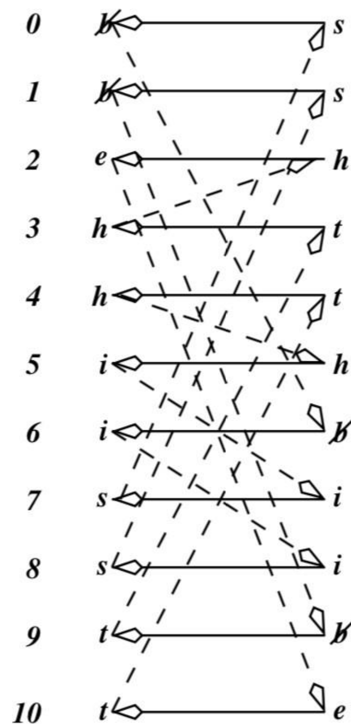
F: b/b/ehhiisstt

We can use L and F to generate the original sequence. Look at Table 6.15 containing the cyclically shifted sequences sorted in lexicographic order. Because each row is a cyclical shift, the letter in the first column of any row is the letter appearing after the last column in the row in the original sequence. If we know that the original sequence is in the kth row, then we can begin unravelling the original sequence starting with the kth element of F.

Example: In our example

$$F = \begin{bmatrix} \not{b} \\ \not{b} \\ e \\ h \\ h \\ i \\ i \\ s \\ s \\ t \\ t \end{bmatrix} \quad L = \begin{bmatrix} s \\ s \\ h \\ t \\ t \\ h \\ \not{b} \\ i \\ i \\ \not{b} \\ e \end{bmatrix}$$

The original sequence is sequence number 10, so the first letter in of the original sequence is F[10] = ***t***. To find the letter following ***t*** we look for ***t*** in the array L. There are two ***t***'s in L. Which should we use? The ***t*** in F that we are working with is the lower of two ***t***'s, so we pick the lower of two ***t***'s in L. This is L[4]. Therefore, the next letter in our reconstructed sequence is F[4] = ***h***. The reconstructed sequence to this point is ***th***. To find the next letter, we look for ***h*** inthe L array. Again, there are two ***h***'s. The ***h*** at F[4] is the lower of two ***h***'s in F, so we pick the lower of the two ***h***'s in L. This is the fifth element of L, so the nextelement in our decoded sequence is F[5] = ***i***. The decoded sequence to this pointis ***thi***. The process continues as depicted in Figure 6.1 to generate the original sequence.



Why go through all this trouble? After all, we are going from a sequence of length ***N*** to another sequence of length ***N*** plus an index value. It appears that we are actually causing expansion instead of compression. The answer is that the sequence L can be compressed much more efficiently than the original sequence. Even in our small example we have runs of like symbols. This will happen a lot more when ***N*** is large. Consider a large sample of text that has been cyclically shifted and sorted. Consider all the rows of A beginning with ***heb/***. With high probability ***heb/*** would be preceded by ***t***. Therefore, in L we would get a long run of ***t***'s.

8.3 Move-to-Front Coding

A coding scheme that takes advantage of long runs of identical symbols is the move-to-front (*MTF*) coding. In this coding scheme, we start with some initial listing of the source alphabet. The symbol at the top of the list is assigned the number 0, the next one is assigned the number 1, and so on. The first time a particular symbol occurs, the number corresponding to its place in the list is transmitted. Then it is moved to the top of the list. If we have a run of this symbol, we transmit a sequence of 0s. This way, long runs of different symbols get transformed to a large number of 0s. Applying this technique to our example does not produce very impressive results due to the small size of the sequence, but we can see how the technique functions.

Example: Let's encode L = sshtthb/iib/e. Let's assume that the source alphabet is given by A = { b/, e, h, i, s, t}.We start out with the assignment

0	1	2	3	4	5
b/	*e*	*h*	*i*	*s*	*t*

The first element of L is s, which gets encoded as a 4. We then move ***s*** to the top of the list, which gives us

0	1	2	3	4	5
s	*b/*	*e*	*h*	*i*	*t*

The next ***s*** is encoded as 0. Because s is already at the top of the list, we do not need to make any changes. The next letter is ***h***, which we encode as 3. We then move ***h*** to the top of the list:

0	1	2	3	4	5
h	*s*	*b/*	*e*	*i*	*t*

The next letter is ***t***, which gets encoded as 5. Moving ***t*** to the top of the list, we get

0	1	2	3	4	5
t	*h*	*s*	*b̸*	*e*	*i*

The next letter is also a ***t***, so that gets encoded as a 0. Continuing in this fashion, we get the sequence:

40350135015

As we warned, the results are not too impressive with this small sequence, but we can see how we would get large numbers of 0s and small values if the sequence to be encoded was longer.